Filtering operation on the GPU - an extension of *ÆminiumGPU [3]*

Maksymilian Wojczuk[1] and Paweł Wacławiak[1]

Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa
{fc54082,fc54101}@alunos.fc.ul.pt

Abstract. The topic of this project is the implementation of filter operation as an extension of java framework ÆminiumGPU that offers an easy execution of code on the Graphic Processing Unit (GPU). Until now the project offered functions *Map* and *Reduce* to execute operations on input data in parallel. These two are the foundation of *Map-Reduce* model, but to be fully functional a filter is needed to sieve-out values that are no longer relevant. Introduction of a filter function creates a wider spectre of possible use cases and data filtration. Our implementation of the filter allows to enqueue multiple instances of filters alternately with mapping functions, reducing the output stream size meanwhile to optimize execution time and memory copying. This project creates an opportunity to easily parallelize any code that aims on transforming and filtering input data and execute it on a GPU efficiently.

Keywords: GPGPU · Filtering · AEminium.

1 Introduction

Since the time, when Graphic Processing Units manufacturers started providing special drivers for programming on the Units primarily designed to graphics processing making it possible to run user-defined code, more and more researchers started to explore the possibilities of using the power of huge number of processing cores. GPUs are an interesting target to program in terms of the possibilities that they provide - namely massive parallelism which is considered the future of processing, however they also introduce many limitations due to their architecture.

Usually, the process of developing software for the GPUs includes using low level APIs provided by manufacturers and also extra knowledge about the architecture of the device itself making it, arguably, a more cumbersome and difficult process. Probably this is the reason why the potential of many GPUs that are in most of the Personal Computers is not fully utilised by the users, especially developers.

For the massive parallelism offered by GPU to be useful the use case usually includes big data processing, algorithms that run for a long time or perform the same operations many times - the overhead of copying the data to the GPU

makes it unsuitable to be used in smaller problems as shown in the [3]. Due to this fact usually it is faster to perform small computations on the CPU.

One of the most common techniques for data processing is the functional inspired model - *mapReduce*. As the name suggests it allows 2 operations, namely *map* and *reduce*. Even though the model seems quite simple it is widely used in data processing [4] and addresses most of the use cases. Apart from those 2 basic operations, probably one more also very needed but much more difficult to implement on the GPU is operation **filter** which allows, as the name suggests filtering - discarding some part of the data basing on a user-defined condition. The goal of the project was to extend the Æminiumgpu framework with a **filter** operation in order to allow users more flexible data processing:

- implementation of the filter operation on the GPU,
- integration with the Æminiumgpu framework making it possible for users to use a high-level abstraction for data processing,
- (optional) optimisation of chained calls minimising the overhead caused by data copying between host and GPU's memory,
- evaluation of the speedup over processing on the CPU gained by such approach depending on processed data size.

2 Background

2.1 ÆminiumGPU framework [3]

The ÆminiumGPU framework is a framework that allows execution of simple operations (Map and Reduce) in parallel on the GPU. It is designed to be used from Java and creates an abstraction over OpenCL code allowing developers to use massive parallel possibilities on the GPU not knowing details of it's architecture. For more details reader should refer to [3].

2.2 Massive Parallel Filtering problem

As already described in the previous section, often the computing power that comes from possibilities of massive parallel programming is compensated with the difficulty of programming part itself. Here we are going to consider a simple function **filter** and it's example implementation on the one-core CPU shown in Listing 1.1

Listing 1.1. Pseudocode of filter function in high level language

```
List filter(List inputList, Function(int => bool) predicate) {
    List outputList = newListOfType(inputList);
    for(element in inputList)
        if(predicate(element)
            outputList.add(element);
    return outputList;
}
```

As one can see, the code is quite simple and intuitive. Even the implementation for simple parallel execution wouldn't be much harder (dividing work into a few threads and then joining lists)[1]. However, on the GPU, where the possibility of running much greater number of threads and using only simple structures such as arrays makes this approach inefficient and almost impossible. To fully understand the typical approach for programming on a GPU we need to think of each process handling only a small amount of elements (or even sometimes just one).

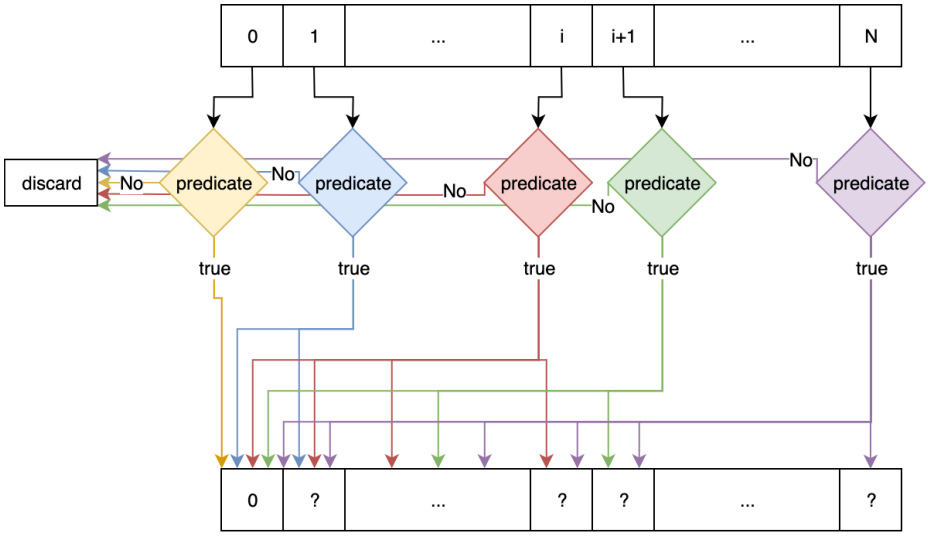

Fig. 1. Filtering each element in a separate thread

Let's imagine, that in the presented example every thread handles only one element. We have an input array which contains elements to be filtered and output array which is an allocated space for the new array containing only filtered elements. Every thread in the kernel call (on the GPU) then processes only one element and passes to the output array. The problem however appears that the index for the new element to be put in the output array is determined by all elements that appear before the processed one - this creates a dependency on all previous elements and makes only a sequential execution possible (element can be processed only when all it's predecessors are already processed). Figure 2.2 shows an example filtering of an array, where every colour stands for a thread processing the one-value input. For the input from index 0 in the array - the case is trivial. If the predicate is true - the value is being copied to the output array under index 0, if no - then it is discarded. For any other value - the output index

[1] of course a truly efficient implementation would require much more work, but this is not the scope of this project

may be in every place from the beginning of the array depending on how many elements have been already placed there.

2.3 Prefix Sum algorithm

In order to understand our approach to the filtering algorithm, it is needed to understand prefix sum algorithm which is a key to parallelization of this filter. Parallel prefix sum algorithm contains of two parts: up-sweep and down-sweep. While the first one is rather simple and intuitive the latter is more complicated and harder to understand.

The role of sweep-up part is calculating partial sums for internal nodes of the array. It is also known as parallel reduce because after this phase the last node of the array holds a sum of all of its nodes.

Down-sweep ensures we get a proper prefix sum array by propagating the calculated values to nodes of lower indexes and overwriting them. At the same time adding the old value to a node of a higher index. After this phase is finished we receive an array starting with zero and containing partial sums up to a corresponding index in each cell.

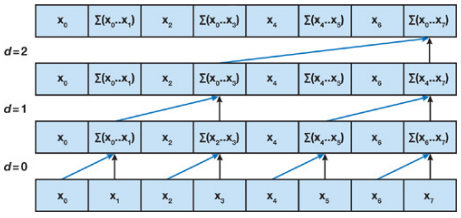

Fig. 2. illustration of up-sweep operation in prefix-sum algorithm

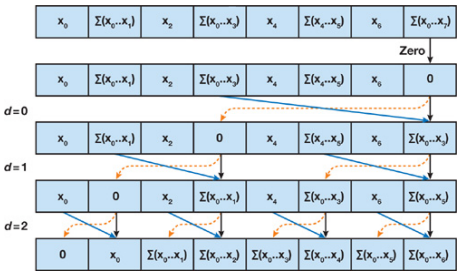

Fig. 3. illustration of down-sweep operation in prefix-sum algorithm

3 Approach

Having the prefix sum algorithm designed for the GPU it is possible to implement a filtering algorithm that uses the result of prefix sum. Here we describe our approach to using this algorithm in a parallel filtering algorithm. Let us imagine, that apart from the input array we also have an extra array of integers (later: predicate array) in which every process applies the predicate and based on the result puts value 1, if the predicate was true and 0 in the other case. In this way we have the predicate array with values determining if the value from the input array should also appear in the output array. The process has been shown on the figure 4.

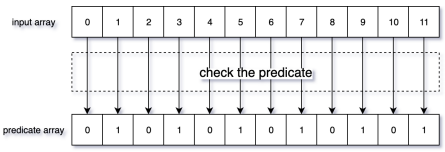

Fig. 4. Checking the predicate and assigning it to the predicate array

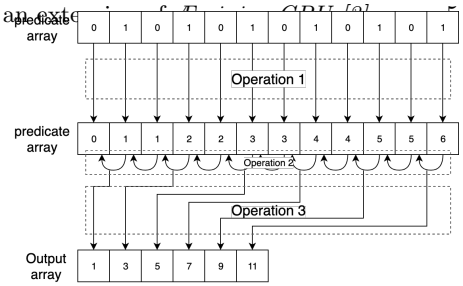

Fig. 5. Operation 1: prefix sum **Operation 2**: for index greater than 0 check if `predicate[index-1] == predicate[index]` **Operation 3**: if operation 2 returns true, then apply `output[predicate[index]-1] = input[index]`

If we now apply the prefix sum algorithm on the predicate array, we will have the value 1 under index of the first element that passed the predicate, value 2 on the second and so on. Having this knowledge of which in turn is every element of the input array passing the predicate we also know where to put it in the output array. The only thing that needs to be clarified is which of the elements having the same value in the predicate array (containing currently prefix sum of original predicate array) is the element, that passed the predicate. However, this is trivial - if the previous element has a smaller value in the predicate array, then this element is the one that needs to be put at the index of value in predicate array - 1. This part of the algorithm has been shown on the figure 5.

4 Implementation

4.1 Implementation details

Filter Algorithm. Our approach could be realized in many ways and in order to shape our implementation to its current form we needed to make some major decisions regarding multi-threading and synchronisation. We have also developed the algorithm using *CUDA* which is more straightforward and then migrated it to *OpenCL*, this also implied some serious issues.

First of them and having the biggest impact on whole structure of the filter was **multi-threading**. First of all we decided to only have half the array length amount of threads. The reason for that is that during the process of calculating the prefix sum only this number of threads make any actions and that is just in the first order of up-sweep and the last one of down-sweep, so half of the threads would not participate in this process at all. The only benefit from having a thread for each cell of input array is halving the time of populating the predicate array and possibly the output array in best case scenario.

Second decision on **threading** and also **synchronisation** was to limit the kernel just to one block of threads. The reason for this was that we have encoun-

tered some difficulties in synchronisation. We have tried implementing a simple semaphore to synchronise blocks. Also coming up with an idea of synchronising only on first thread of each block to optimize the process and inside each block with *CLK_MEM_GLOBAL_FENCE* built-in OpenCL function, but this solution was sometimes producing unpredictable outcomes. It also had some drawbacks like atomic operation overhead with bigger number of blocks and was not supported in some older versions of OpenCL. There is also a limitation that could cause problems sometimes, namely even though a GPU has a certain number of cores and can form a number of groups of these cores one kernel can be limited to a certain number of working-groups. We are aware that our approach on this topic is not the most efficient and a combination of this and multi-work-group synchronisation could be a way of optimization.

Third decision of ours was to reduce the amount of used threads as much as possible with preserving the same level of throughput. The standard way of allocating work-units would be to compare number of needed work-units and the maximum size of work-block and choose the bigger one. In case we needed work-units number equal to 110% of maximum work-group size, while computing values for the last 10% of the array 90% of work-units would be just waiting for the others to finish the calculations and that is an apparent waste of resources. Instead of that we are checking how many work-groups would we need to calculate everything in parallel and divide the number of needed work-units by this number, that gives us the optimal number of threads executing at the same time. With this optimisation it is possible to reduce number of used GPU cores up to 50% what lets more programs execute at the same time.

Æminium integration. The real power of **Map-Filter-Reduce** is chaining multiple operations in order to get the desired output. We managed to implement **Filter-Filter** chaining as merging predicates in a single kernel and also **Filter-Reduce** that applies reduce function only on elements that pass the predicate in the same kernel.

Map-Filter combinations would be more profitable in terms of execution effectiveness but are also more complicated to implement. The problems that we faced in the implementation of the Map-Filter combinations is that we cannot easily merge code in the kernel - possibilities of combinations of those 2 operations are infinite and it is not possible to handle them in the same way. Also, since Filter operations change the number of elements in the array it is sometimes unpractical to chain multiple map-filter operations in one kernel using the same number of threads (e.g. let's imagine that the first filter operation reduces the number of elements in the input array by 90% - there is no need to compute map operation on all elements, but just on the result from the filter operation).

The idea, that seemed the best to handle chaining of multiple operations was introduction of `Lazy Lists`, that would allow separate kernel calls but on the same allocated buffer on the GPUs memory without copying it to the host's memory. Such lists would allow users to operate on abstraction, the actual values would be copied to the host's memory on user request (e.g. getting an element).

In this way it would be possible to minimise the overhead connected to memory copying in multiple chained operations in the same time mitigating the problem of infinite number of chained combinations and too many threads operating on a filtered table. Unfortunately we haven't managed to implement this solution, but certainly this could be a Future Work.

4.2 Problems

We have encountered two main problems in the last stage of implementation process having the code already migrated to *OpenCL*. Even though the kernel code structure was the same for both versions it did not work with whole Æminiumgpu project. It gave us a hard time debugging.

First of them was related to work-item index. It appeared that the data type returned by OpenCL's function *get_global_id()* is *size_t* which is not compatible with *integer* we were using to store the value and all the work-items were ending up with *id = 0*.

The second one was hidden much deeper and much harder to find. We knew that the GPU we were using for testing supports work-groups of max size 1024, this piece of information was gathered dynamically from the system. It turned out that our kernel uses a maximum number of work-items equal to 256 even when it is expected to use more. Therefore we infer that in *OpenCL 1.2* there is an additional limitation that only allows to use 256 work-items per group.

5 Evaluation

5.1 Experimental Setup

Listing 1.2. Testing CPU details

```
Model name:            Intel(R) Xeon(R) CPU X5670  @ 2.93GHz
Core(s) per socket:    6
Thread(s) per core:    2
CPU MHz:               1598.388
CPU max MHz:           2927
CPU(s):                24
```

Listing 1.3. Testing GPU details

```
Device name:                    GeForce GTX 960
Max clock frequency:            1291 (?)
Global memory size:             4235919360 (4 GB)
Max memory allocation size:     1058979840 (1 GB)
Max work group size:            1024
Max compute units:              8 (max working groups per kernel)
OpenCL C version:               OpenCL C 1.2
```

5.2 Results

We have run the project on test data sets of different sizes and compared the execution times with parallel CPU version and sequential CPU version. Results are shown on the figure 5.2. We have run different benchmarks with different input size (10, 100, 1e3, 1e4, 1e5, 1e6, 1e7, 3e7, 4e7) performing a simple operation of filtering even numbers (integers). Every test was ran 10 times and average of all test has been taken as the result.

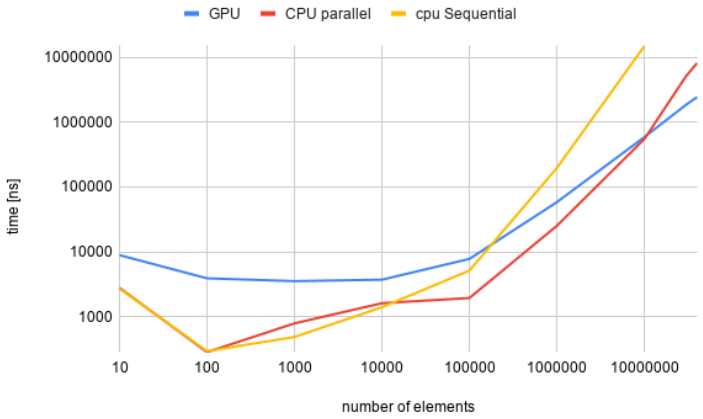

Fig. 6. GPU vs CPU time comparison

5.3 Discussion

As one can infer from the graphs using this approach is profitable mostly for long input arrays. The complexity for this solution equals to

```
O(N/T+log(N)), T - number of threads (work-items).
```

The more threads we can use the faster the program is going to be. That implies that implementation which utilizes multiple work-groups would be the desired even thought it introduces some overhead on synchronisation.

The overhead of calculating prefix sum in parallel is big enough to make the parallel GPU version lose with sequential CPU version for arrays of lower number of elements, but with bigger number of elements to filter time advantage of parallel version is going to grow gradually.

While testing and debugging the code we have found out, that when using OpenCL, the maximum number of work-units in a work-group is only 256 what slows down the execution 4 times in comparison to NVIDIA CUDA, where we could obtain a block of size 1024. Because of this limitation our solution is more limited and does not use fully possibilities of the GPU. Probably also because of this, computation on the GPU was superior over CPU only on data of size bigger than 10000000.

6 Related Work

There are already many frameworks on the market targeting a facilitation of GPU programming - creation of an abstraction layer allowing developers to flawlessly use the power offered by GPUs not having to care about it's specific architecture [2, 6, 10, 13].

There is also a lot of work done in the area of data processing using basic operations such as Map, Filter and Reduce which seem able to solve many common data processing problems [5, 8, 11].

There has also been a lot of research done in efficient algorithms allowing massive parallel processing that depends on the data appearing under preceding indexes in an array - scanning algorithm of which the algorithm that we used (prefix sum) is a specific case [12].

One of the most worth mentioning solutions is a relatively recent but very fast-growing Open Source project Rapids [9] incubated by NVIDIA directed to high level data processing using GPUs. This solution requires more attention since it is a whole ecosystem targeting GPU programming. It relies on numba [7] and apache arrow [1].

7 Conclusions

We have implemented operation *filter* for GPU that uses prefix-sum algorithm. As shown in the 5.2 for certain amount of data it is more efficient than running operations on the CPU. For small amount of data the CPU implementation is more efficient, but the overall solution is designed for bigger data processing.

We believe that there still are some improvements that were outside of the scope of this project. Namely - we haven't tested work splitting between the CPU and GPU in filter operation - while the GPU is processing part of the data, the CPU could be also working processing the rest - finding an ideal split point that allows work division could be implemented in the feature.

Also, a crucial part of the *Map-Filter-Reduce* model is operation chaining. While our solution does not implement any optimisation of chaining Map-Filter operations this could be another idea for future work within Æminium GPU framework.

Acknowledgements

First Author focused on integrating the Æminiumgpu framework with the kernel code for operation *filter*. Second Author implemented the kernel code for filtering.

Both authors wrote this paper, with First Author focusing on the introduction, Background (2.1, 2.2), Approach, related work and conclusions while the Second Author focused on abstract, background (2.3), implementation and evaluation.

Each author spent around 40 hours on this project.

References

1. Apache: Apache Arrow. https://arrow.apache.org/
2. Clarkson, J., Kotselidis, C., Brown, G., Luján, M.: Boosting java performance using gpgpus. In: Knoop, J., Karl, W., Schulz, M., Inoue, K., Pionteck, T. (eds.) Architecture of Computing Systems - ARCS 2017 - 30th International Conference, Vienna, Austria, April 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10172, pp. 59–70. Springer (2017). https://doi.org/10.1007/978-3-319-54999-6_5, https://doi.org/10.1007/978-3-319-54999-6_5
3. Fonseca, A., Cabral, B.: Æminiumgpu: An intelligent framework for GPU programming. In: Keller, R., Kramer, D., Weiss, J. (eds.) Facing the Multicore-Challenge - Aspects of New Paradigms and Technologies in Parallel Computing [Proceedings of a conference held at Stuttgart, Germany, September 19-21, 2012]. Lecture Notes in Computer Science, vol. 7686, pp. 96–107. Springer (2012). https://doi.org/10.1007/978-3-642-35893-7_9, https://doi.org/10.1007/978-3-642-35893-7_9
4. Hadoop, A.: Apache hadoop. URL http://hadoop. apache. org (2011)
5. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A mapreduce framework on graphics processors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. p. 260–269. PACT '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1454115.1454152, https://doi.org/10.1145/1454115.1454152
6. Ishizaki, K., Hayashi, A., Koblents, G., Sarkar, V.: Compiling and optimizing java 8 programs for GPU execution. In: 2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015. pp. 419–431. IEEE Computer Society (2015). https://doi.org/10.1109/PACT.2015.46, https://doi.org/10.1109/PACT.2015.46
7. Lam, S.K., Pitrou, A., Seibert, S.: Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. LLVM '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2833157.2833162, https://doi.org/10.1145/2833157.2833162
8. Lee, S., Chakravarty, M.M., Grover, V., Keller, G.: Gpu kernels as data-parallel array computations in haskell. In: Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods. pp. 1–9 (2009)
9. Nvidia: Nvidia Rapids. https://rapids.ai/about.html, [https://github.com/rapidsai]
10. Nystrom, N., White, D., Das, K.: Firepile: Run-time compilation for gpus in scala. SIGPLAN Not. **47**(3), 107–116 (Oct 2011). https://doi.org/10.1145/2189751.2047883, https://doi.org/10.1145/2189751.2047883
11. Rajanna, K., Das, K.: Gpu parallel collections for scala (2011)
12. Sengupta, S., Harris, M.J., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Segal, M., Aila, T. (eds.) Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007, San Diego, California, USA, August 4-5, 2007. pp. 97–106. Eurographics Association (2007). https://doi.org/10.2312/EGGH/EGGH07/097-106, https://doi.org/10.2312/EGGH/EGGH07/097-106

13. Yan, Y., Grossman, M., Sarkar, V.: JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In: Sips, H.J., Epema, D.H.J., Lin, H. (eds.) Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5704, pp. 887–899. Springer (2009). https://doi.org/10.1007/978-3-642-03869-3_82, https://doi.org/10.1007/978-3-642-03869-3_82